Tensorflow 代码解析

1.TF 系统架构

1.1 TF 依赖视图

TF 的依赖视图如图 1.1 所示，描述了 TF 的上下游关系链。

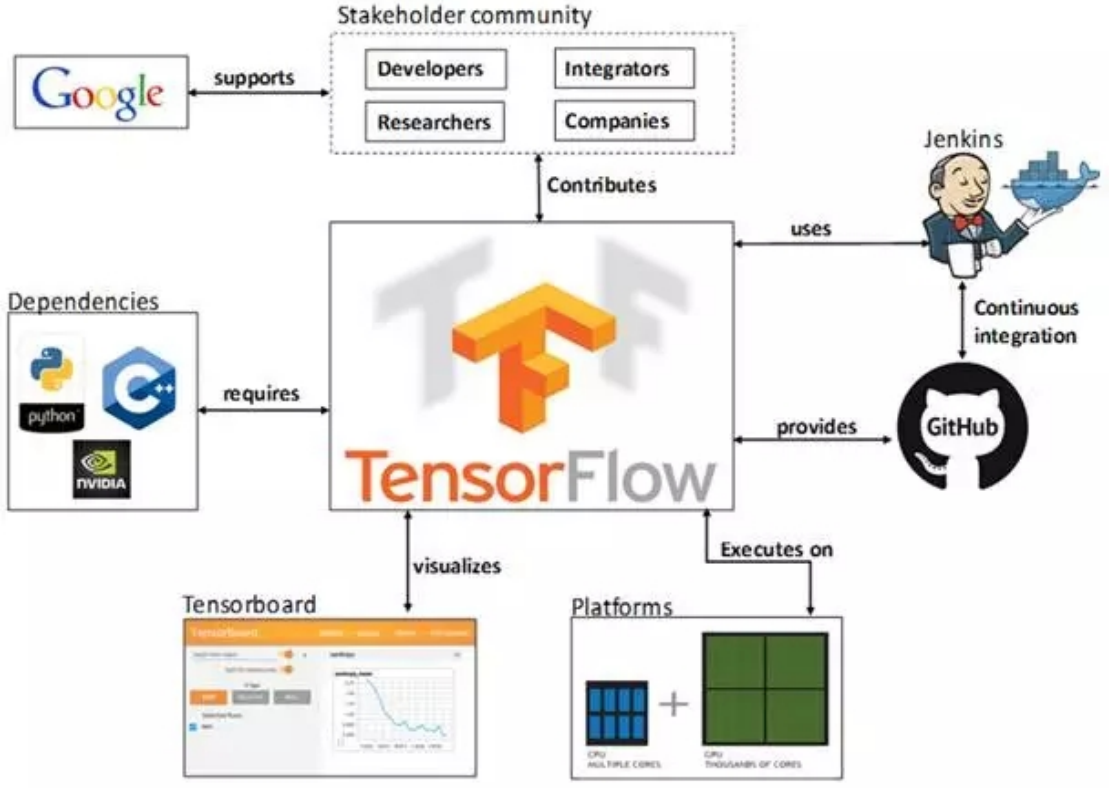

图 1.1

TF 托管在 github 平台，有 google groups 和 contributors 共同维护。
TF 提供了丰富的深度学习相关的 API，支持 Python 和 C/C++接口。
TF 提供了可视化分析工具 Tensorboard，方便分析和调整模型。
TF 支持 Linux 平台，Windows 平台，Mac 平台，甚至手机移动设备等各种平台。

1.2TF 系统架构

图 1.2 是 TF 的系统架构，从底向上分为设备管理和通信层、数据操作层、图计算层、API 接口层、应用层。其中设备管理和通信层、数据操作层、图计算层是 TF 的核心层。

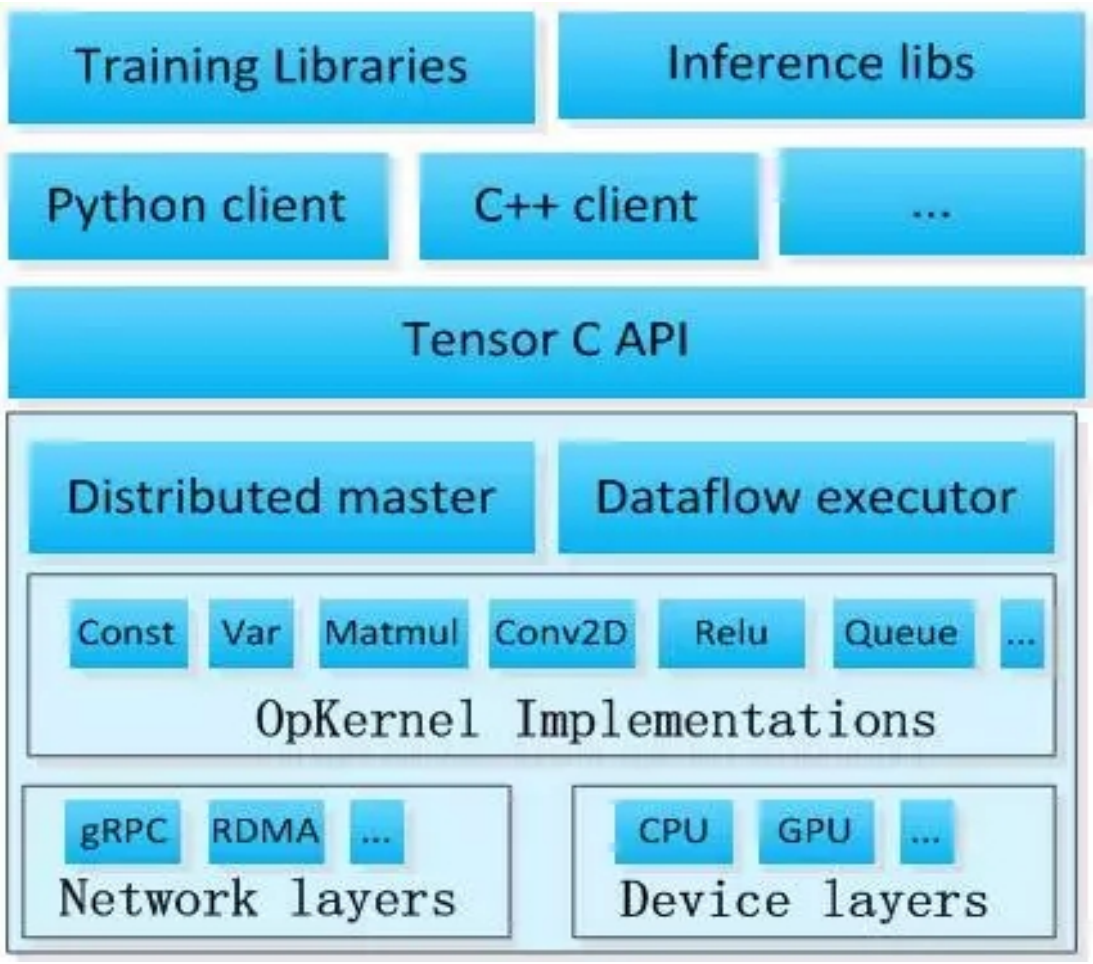

图 1.2TF 系统架构

- 底层设备通信层负责网络通信和设备管理。设备管理可以实现 TF 设备异构的特性，支持 CPU、GPU、Mobile 等不同设备。网络通信依赖 gRPC 通信协议实现不同设备间的数据传输和更新。
- 第二层是 Tensor 的 OpKernels 实现。这些 OpKernels 以 Tensor 为处理对象，依赖网络通信和设备内存分配，实现了各种 Tensor 操作或计算。Opkernels 不仅包含 MatMul 等计算操作，还包含 Queue 等非计算操作，这些将在第 5 章 Kernels 模块详细介绍。
- 第三层是图计算层（Graph），包含本地计算流图和分布式计算流图的实现。Graph 模块包含 Graph 的创建、编译、优化和执行等部分，Graph 中每个节点都是 OpKernels 类型表示。关于图计算将在第 6 章 Graph 模块详细介绍。
- 第四层是 API 接口层。Tensor C API 是对 TF 功能模块的接口封装，便于其他语言平台调用。
- 第四层以上是应用层。不同编程语言在应用层通过 API 接口层调用 TF 核心功能实现相关实验和应用。

1.3TF 代码目录组织

图 1.3 是 TF 的代码结构视图，下面简单介绍 TF 的目录组织结构。

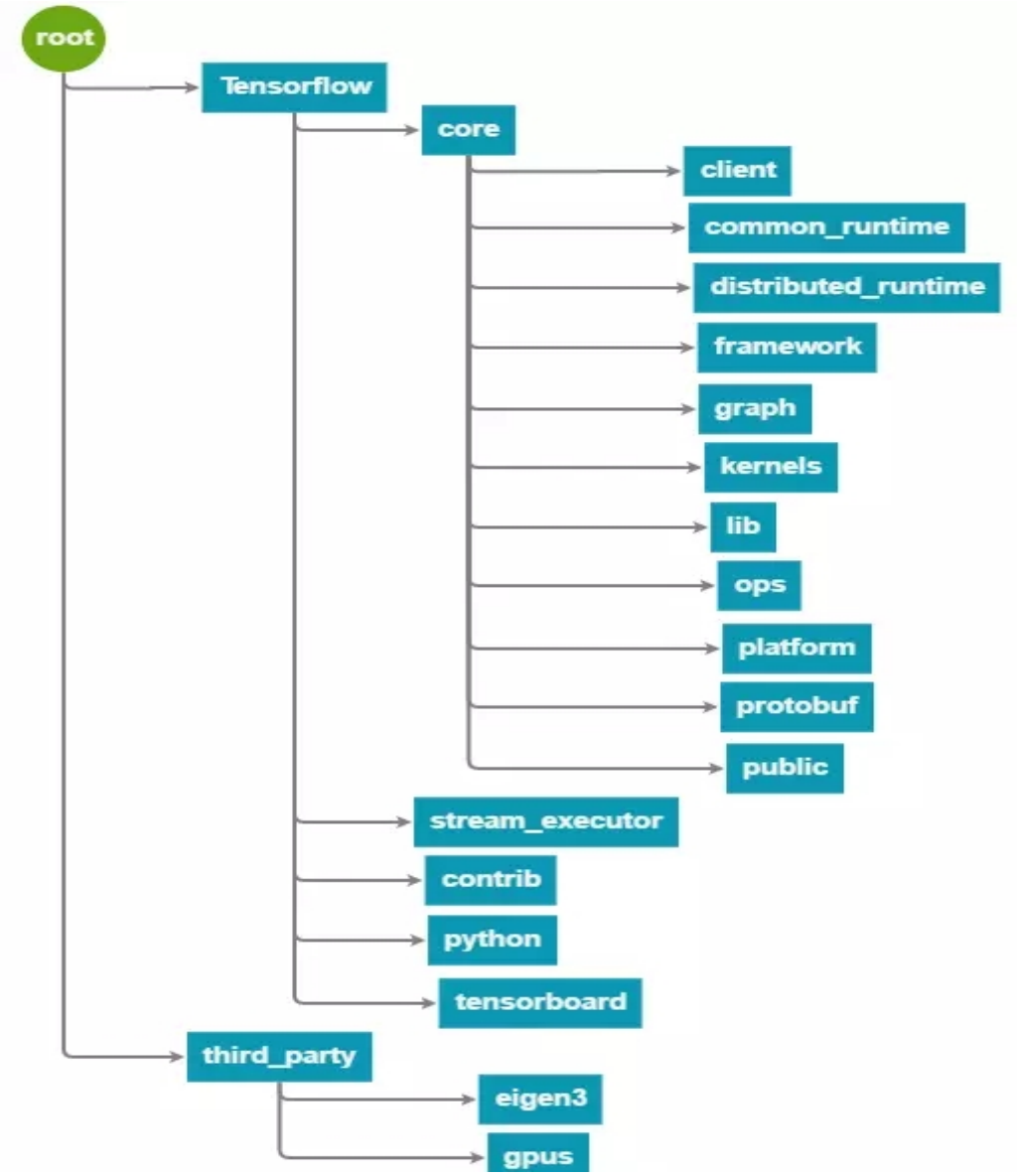

图 1.3 代码目录结构

Tensorflow/core 目录包含了 TF 核心模块代码。

public: API 接口头文件目录，用于外部接口调用的 API 定义，主要是 session.h 和 tensor_c_api.h。

client: API 接口实现文件目录。

platform: OS 系统相关接口文件，如 file system, env 等。

protobuf: 均为.proto 文件，用于数据传输时的结构序列化.

common_runtime: 公共运行库，包含 session, executor, threadpool, rendezvous, memory 管理, 设备分配算法等。

distributed_runtime: 分布式执行模块，如 rpc session, rpc master, rpc worker, graph manager。

framework: 包含基础功能模块，如 log, memory, tensor

graph: 计算流图相关操作，如 construct, partition, optimize, execute 等

kernels: 核心 Op，如 matmul, conv2d, argmax, batch_norm 等

lib: 公共基础库，如 gif、gtl(google 模板库)、hash、histogram 等。

ops: 基本 ops 运算，ops 梯度运算，io 相关的 ops，控制流和数据流操作

Tensorflow/stream_executor 目录是并行计算框架，由 google stream executor 团队开发。

Tensorflow/contrib 目录是 contributor 开发目录。

Tensroflow/python 目录是 python API 客户端脚本。

Tensorflow/tensorboard 目录是可视化分析工具，不仅可以模型可视化，还可以监控模型参数变化。

third_party 目录是 TF 第三方依赖库。

eigen3: eigen 矩阵运算库，TF 基础 ops 调用

gpus: 封装了 cuda/cudnn 编程库

2. TF 核心概念

TF 的核心是围绕 Graph 展开的，简而言之，就是 Tensor 沿着 Graph 传递闭包完成 Flow 的过程。所以在介绍 Graph 之前需要讲述一下符号编程、计算流图、梯度计算、控制流的概念。

2.1 Tensor

在数学上，Matrix 表示二维线性映射，Tensor 表示多维线性映射，Tensor 是对 Matrix 的泛化，可以表示 1-dim、2-dim、N-dim 的高维空间。图 2.1 对比了矩阵乘法（Matrix Product）和张量积（Tensor Contract），可以看出 Tensor 的泛化能力，其中张量积运算在 TF 的 MatMul 和 Conv2D 运算中都有用到，

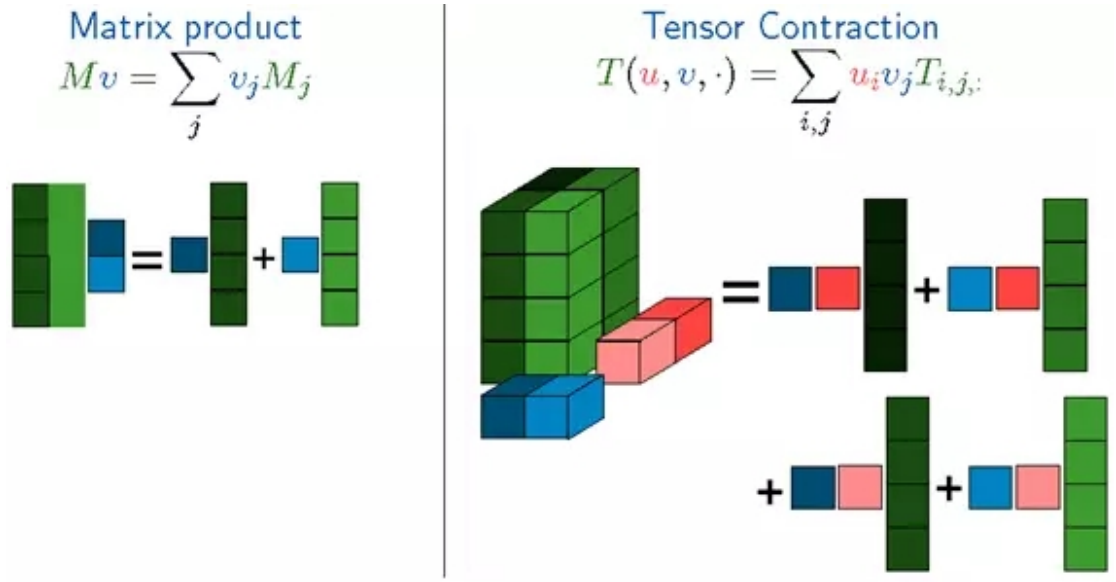

图 2.1 Tensor Contraction

Tensor 在高维空间数学运算比 Matrix 计算复杂，计算量也非常大，加速张量并行运算是 TF 优先考虑的问题，如 add, contract, slice, reshape, reduce, shuffle 等运算。

TF 中 Tensor 的维数描述为阶，数值是 0 阶，向量是 1 阶，矩阵是 2 阶，以此类推，可以表示 n 阶高维数据。

TF 中 Tensor 支持的数据类型有很多，如 tf.float16, tf.float32, tf.float64, tf.uint8, tf.int8, tf.int16, tf.int32, tf.int64, tf.string, tf.bool, tf.complex64 等，所有 Tensor 运算都使用泛化的数据类型表示。

TF 的 Tensor 定义和运算主要是调用 Eigen 矩阵计算库完成的。TF 中 Tensor 的 UML 定义如图 2.2。其中 TensorBuffer 指针指向 Eigen::Tensor 类型。其中，Eigen::Tensor 不属于 Eigen 官方维护的程序，由贡献者提供文档和维护，所以 Tensor 定义在 Eigen unsupported 模块中。

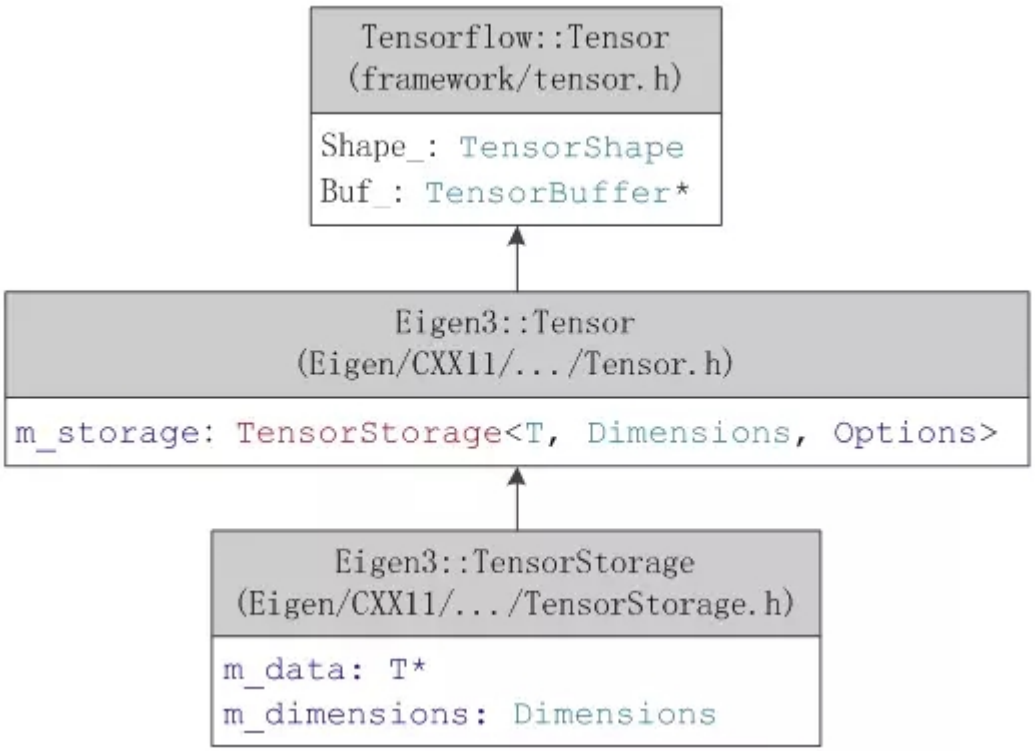

图 2.2 Tensor 数据结构定义

图 2.2 中，Tensor 主要包含两个变量 m_data 和 m_dimension，m_data 保存了 Tensor 的数据块，T 是泛化的数据类型，m_dimensions 保存了 Tensor 的维度信息。

Eigen::Tensor 的成员变量很简单，却支持非常多的基本运算，再借助 Eigen 的加速机制实现快速计算，参考章节 3.2。Eigen::Tensor 主要包含:

一元运算（Unary），如 sqrt、square、exp、abs 等。

二元运算（Binary），如 add，sub，mul，div 等

选择运算（Selection），即 if / else 条件运算

归纳运算（Reduce），如 reduce_sum， reduce_mean 等

几何运算（Geometry），如 reshape，slice，shuffle，chip，reverse，pad，concatenate，extract_patches，extract_image_patches 等

张量积（Contract）和卷积运算（Convolve）是重点运算。

2.2 符号编程

编程模式通常分为命令式编程（imperative style programs）和符号式编程（symbolic style programs）。

命令式编程容易理解和调试，命令语句基本没有优化，按原有逻辑执行。符号式编程涉及较多的嵌入和优化，不容易理解和调试，但运行速度有同比提升。

这两种编程模式在实际中都有应用，Torch 是典型的命令式风格，caffe、theano、mxnet 和 Tensorflow 都使用了符号式编程。其中 caffe、mxnet 采用了两种编程模式混合的方法，而 Tensorflow 是完全采用了符号式编程，Theano 和 Tensorflow 的编程模式更相近。

命令式编程是常见的编程模式，编程语言如 python/C++都采用命令式编程。命令式编程明确输入变量，并根据程序逻辑逐步运算，这种模式非常在调试程序时进行单步跟踪，分析中间变量。举例来说，设 A=10, B=10，计算逻辑

```
C=A*B
D=C+1
```

第一步计算得出 C=100，第二步计算得出 D=101，输出结果 D=101。

符号式编程将计算过程抽象为计算图，计算流图可以方便的描述计算过程，所有输入节点、运算节点、输出节点均符号化处理。计算图通过建立输入节点到输出节点的传递闭包，从输入节点出发，沿着传递闭包完成数值计算和数据流动，直到达到输出节点。这个过程经过计算图优化，以数据（计算）流方式完成，节省内存空间使用，计算速度快，但不适合程序调试，通常不用于编程语言中。举上面的例子，先根据计算逻辑编写符号式程序并生成计算图

```
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
F = compile(D)
```

其中 A 和 B 是输入符号变量，C 和 D 是运算符号变量，compile 函数生成计算图 F，如图 2.3 所示。

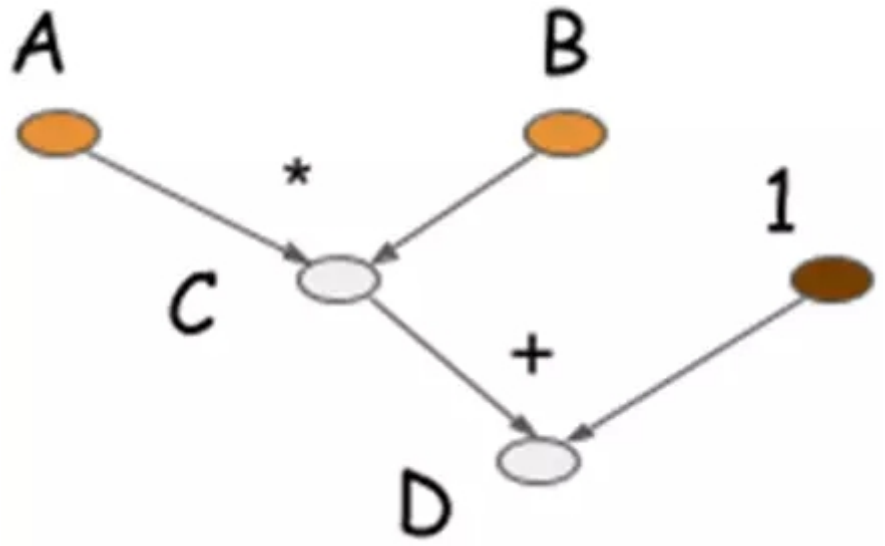

图 2.3 符号编程的正向计算图

最后得到 A=10, B=10 时变量 D 的值，这里 D 可以复用 C 的内存空间，省去了中间变量的空间存储。

D=F(A=10, B=10)

图 2.4 是 TF 中的计算流图，C=F(Relu(Add(MatMul(W, x), b)))，其中每个节点都是符号化表示的。通过 session 创建 graph，在调用 session.run 执行计算。

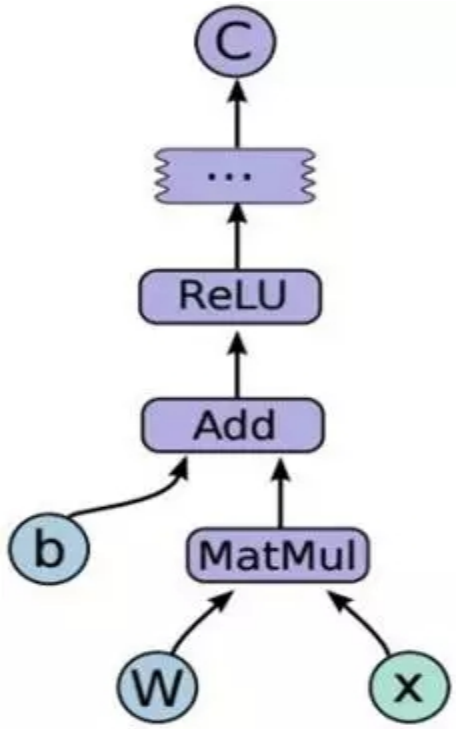

图 2.4 TF 符号计算图

和目前的符号语言比起来，TF 较大的特点是强化了数据流图，引入了 mutation 的概念。这一点是 TF 和包括 Theano 在内的符号编程框架较大的不同。所谓 mutation，就是可以在计算的过程更改一个变量的值，而这个变量在计算的过程中会被带入到下一轮迭代里面去。

Mutation 是机器学习优化算法几乎必须要引入的东西（虽然也可以通过 immutable replacement 来代替，但是会有效率的问题）。 Theano 的做法是引入了 update statement 来处理 mutation。TF 选择了纯符号计算的路线，并且直接把更新引入了数据流图中去。从目前的白皮书看还会支持条件和循环。这样就几乎让 TF 本身成为一门独立的语言。不过这一点会导致最后的 API 设计和使用需要特别小心，把 mutation 引入到数据流图中会带来一些新的问题，比如如何处理写与写之间的依赖。

2.3 梯度计算

梯度计算主要应用在误差反向传播和数据更新，是深度学习平台要解决的核心问题。梯度计算涉及每个计算节点，每个自定义的前向计算图都包含一个隐式的反向计算图。从数据流向上看，正向计算图是数据从输入节点到输出节点的流向过程，反向计算图是数据从输出节点到输入节点的流向过程。

图 2.5 是 2.2 节中图 2.3 对应的反向计算图。图中，由于 C=A*B，则 dA=B*dC, dB=A*dC。在反向计算图中，输入节点 dD，输出节点 dA 和 dB，计算表达式为 dA=B*dC=B*dD,

dB=A*dC=A*dD。每一个正向计算节点对应一个隐式梯度计算节点。

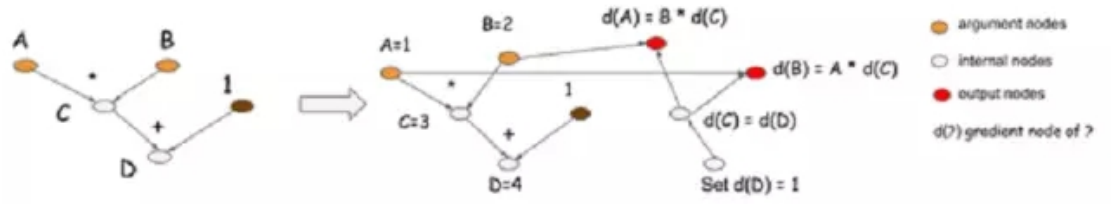

图 2.5 符号编程反向传播

反向计算限制了符号编程中内存空间复用的优势，因为在正向计算中的计算数据在反向计算中也可能要用到。从这一点上讲，粗粒度的计算节点比细粒度的计算节点更有优势，而 TF 大部分为细粒度操作，虽然灵活性很强，但细粒度操作涉及到更多的优化方案，在工程实现上开销较大，不及粗粒度简单直接。在神经网络模型中，TF 将逐步侧重粗粒度运算。

2.4 控制流

TF 的计算图如同数据流一样，数据流向表示计算过程，如图 2.6。数据流图可以很好的表达计算过程，为了扩展 TF 的表达能力，TF 中引入控制流。

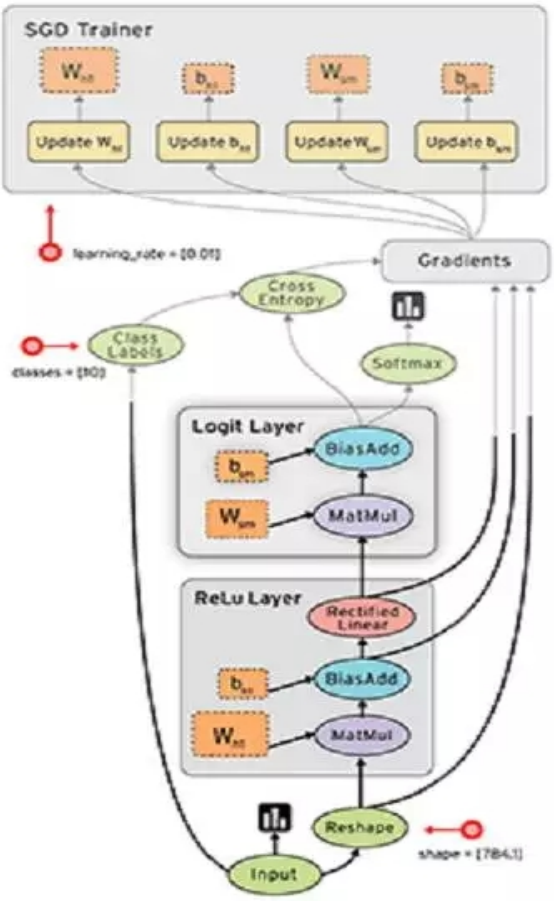

图 2.6 Graph 的数据流

在编程语言中，if…else…是最常见的逻辑控制，在 TF 的数据流中也可以通过这种方式控制数据流向。接口函数如下，pred 为判别表达式，fn1 和 fn2 为运算表达式。当 pred 为 true 是，执行 fn1 操作；当 pred 为 false 时，执行 fn2 操作。

```
tf.cond(pred, fn1, fn2, name=None)
```

TF 还可以协调多个数据流，在存在依赖节点的场景下非常有用，例如节点 B 要读取模型参数 θ 更新后的值，而节点 A 负责更新参数 θ，则节点 B 必须等节点 A 完成后才能执行，否则读取的参数 θ 为更新前的数值，这时需要一个运算控制器。接口函数如下，
tf.control_dependencies 函数可以控制多个数据流执行完成后才能执行接下来的操作，通常与 tf.group 函数结合使用。

```
tf.control_dependencies(control_inputs)
```

TF 支持的控制算子有 Switch、Merge、Enter、Leave 和 NextIteration 等。

TF 不仅支持逻辑控制，还支持循环控制。TF 使用和 MIT Token-Tagged machine 相似的表示系统，将循环的每次迭代标记为一个 tag，迭代的执行状态标记为一个 frame，但迭代所需的数据准备好的时候，就可以开始计算，从而多个迭代可以同时执行。

3. TF 代码分析初步

3.1 Eigen 介绍

在 Tensoflow 中核心数据结构和运算主要依赖于 Eigen 和 Stream Executor 库，其中 Eigen 支持 CPU 和 GPU 加速计算，Stream Executor 主要用于 GPU 环境加速计算。下面简单讲述 Eigen 库的相关特性，有助于进一步理解 Tensorflow。

3.1.1 Eigen 简述

Eigen 是高效易用的 C++开源库，有效支持线性代数，矩阵和矢量运算，数值分析及其相关的算法。不依赖于任何其他依赖包，安装使用都很简便。具有如下特性：

- 支持整数、浮点数、复数，使用模板编程，可以为特殊的数据结构提供矩阵操作。比如在用 ceres-solver 进行做优化问题（比如 bundle adjustment）的时候，有时候需要用模板编程写一个目标函数，ceres 可以将模板自动替换为内部的一个可以自动求微分的特殊的 double 类型。而如果要在这个模板函数中进行矩阵计算，使用 Eigen 就会非常方便。
- 支持逐元素、分块、和整体的矩阵操作。
- 内含大量矩阵分解算法包括 LU，LDLt，QR、SVD 等等。
- 支持使用 Intel MKL 加速
- 部分功能支持多线程
- 稀疏矩阵支持良好，到今年新出的 Eigen3.2，已经自带了 SparseLU、SparseQR、共轭梯度（ConjugateGradient solver）、bi conjugate gradient stabilized solver 等解稀疏矩阵的功能。同时提供 SPQR、UmfPack 等外部稀疏矩阵库的接口。
- 支持常用几何运算，包括旋转矩阵、四元数、矩阵变换、AngleAxis（欧拉角与 Rodrigues 变换）等等。
- 更新活跃，用户众多（Google、WilliowGarage 也在用），使用 Eigen 的比较著名的开

源项目有 ROS（机器人操作系统）、PCL（点云处理库）、Google Ceres（优化算法）。OpenCV 自带到 Eigen 的接口。

- Eigen 库包含 Eigen 模块和 unsupported 模块，其中 Eigen 模块为 official module，unsupported 模块为开源贡献者开发的。

Eigen unsupported 模块中定义了数据类型 Tensor 及相关函数，包括 Tensor 的存储格式，Tensor 的符号表示，Tensor 的编译加速，Tensor 的一元运算、二元运算、高维度泛化矩阵运算，Tensor 的表达式计算。本章后续所述 Tensor 均为 Eigen::Tensor

Eigen 运算性能评估如图 3.1 所示，eigen3 的整体性能比 eigen2 有很大提升，与 GOTO2、INTEL_MKL 基本持平。

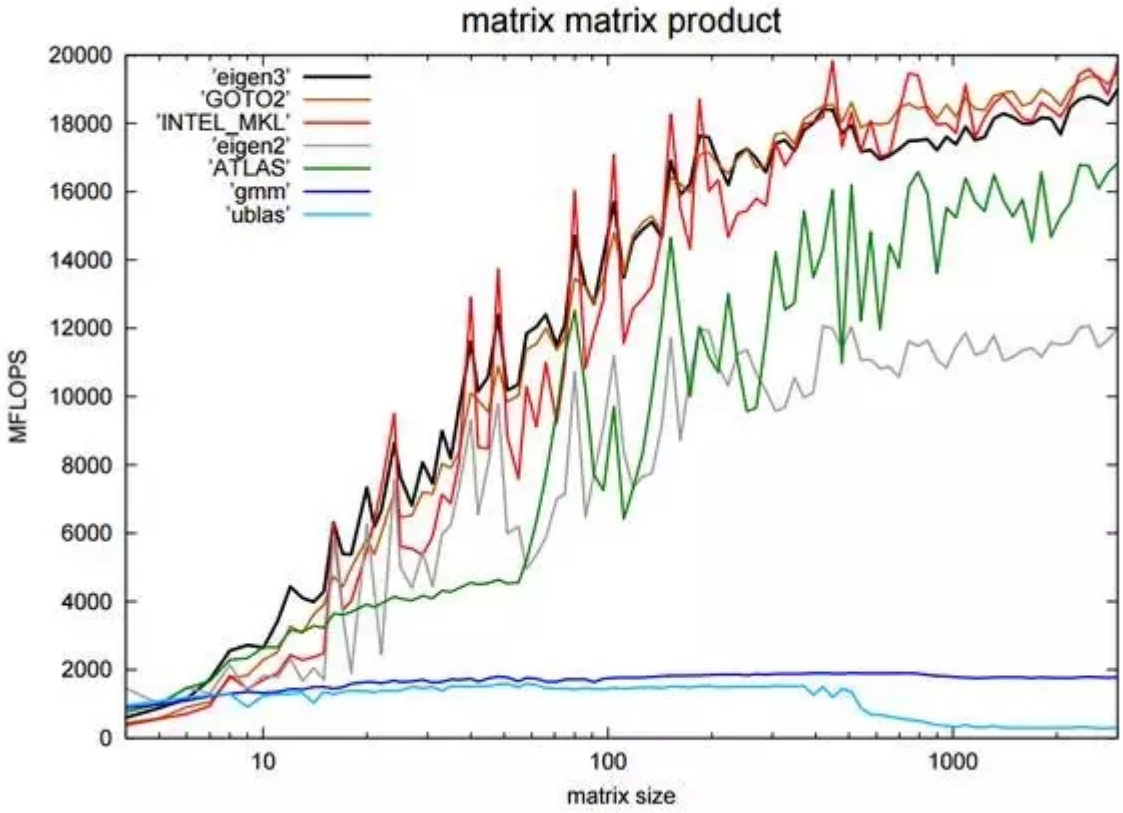

图 3.1 矩阵运算常用库比较

3.1.2 Eigen 存储顺序

Eigen 中的 Tensor 支持两种存储方式:

- Row-major 表示矩阵存储时按照 row-by-row 的方式。
- Col-major 表示矩阵存储时按照 column-by-column 的方式。

Eigen 默认采用 Col-major 格式存储的（虽然也支持 Row-major，但不推荐），具体采用什么存储方式取决于算法本身是行遍历还是列遍历为主。例如：A=[[a11, a12, a13], [a21, a22, a23]]的存储序列见图 3.2。

Row-major order,
e.g., C

Address	Value
0	a_{11}
1	a_{12}
2	a_{13}
3	a_{21}
4	a_{22}
5	a_{23}

Column-major order,
e.g., Fortran

Address	Value
0	a_{11}
1	a_{21}
2	a_{12}
3	a_{22}
4	a_{13}
5	a_{23}

图 3.2 Row-major 和 Column-major 存储顺序

3.1.3 Eigen 惰性求值

在编程语言理论中，存在及早求值(Eager Evaluation) 和惰性求值（Lazy Evaluation）

- 及早求值：大多数编程语言所拥有的普通计算方式
- 惰性求值：也认为是“延迟求值”，可以提高计算性能，最重要的好处是它可以构造一个无限的数据类型。

关于惰性求值，举例如下：

```
Vec3 = vec1 + vec2;
```

及早求值形式需要临时变量 vec_temp 存储运算结果，再赋值给 vec3，计算效率和空间效率都不高：

```
Vec_temp = vec1 + vec2;
Vec3 = vec_temp
```

而惰性求值不需要临时变量保存中间结果，提高了计算性能：

```
Vec_symbol_3 = (vec_symbol_1 + vec_symbol_2);
Vec3 = vec_symbol_3.eval(vec1, vec2)
```

由于 Eigen 默认采用惰性计算，如果要求表达式的值可以使用 Tensor::eval()函数。
Tensor::eval()函数也是 session.run()的底层运算。例如：

```
Tensor<float, 3> result = ((t1 + t2).eval() * 0.2f).exp();
```

3.1.4 Eigen 编译加速

编译加速可以充分发挥计算机的并行计算能力，提高程序运行速度。
举例如下：
普通的循环相加运算时间复杂度是 O(n)：

```
for(int i = 0; i < size; i++) u[i] = v[i] + w[i];
```

如果指令集支持 128bit 并行计算，则时间复杂度可缩短为 O(n/4)：

```
for(int i = 0; i < 4*(size/4); i+=4)
    u.packet(i) = v.packet(i) + w.packet(i);
```

Eigen 编译时使用了 SSE2 加速。假设处理 float32 类型，指令集支持 128bit 并行计算，则一次可以计算 4 个 float32 类型，速度提升 4 倍。

3.1.5 Eigen::half

Tensorflow 支持的浮点数类型有 float16, float32, float64，其中 float16 本质上是 Eigen::half 类型，即半精度浮点数。关于半精度浮点数，英伟达 2002 年首次提出使用半精度浮点数达到降低数据传输和存储成本的目的。
在分布式计算中，如果对数据精度要求不那么高，可以将传输数据转换为 float16 类型，这样可以大大缩短设备间的数据传输时间。在 GPU 运算中，float16 还可以减少一般的内存占用。
在 Tensorflow 的分布式传输中，默认会将 float32 转换为 float16 类型。Tensorflow 的转换方式不同于 nvidia 的标准，采用直接截断尾数的方式转化为半精度浮点数，以减少转换时间。
图 3.3 是双精度浮点数(float64)存储格式。

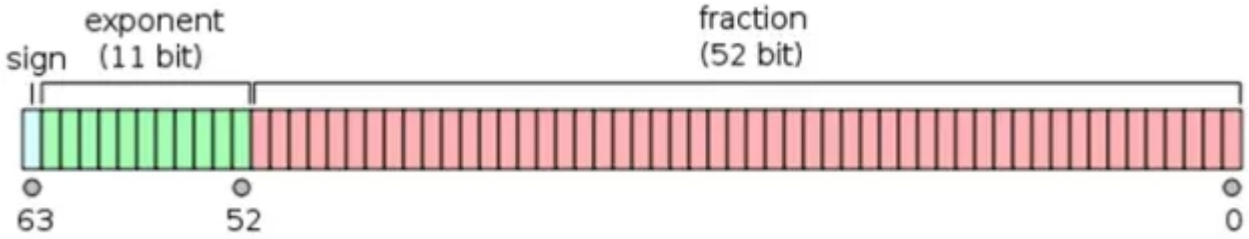

图 3.3 双精度浮点数

图 3.4 是单精度浮点数(float32)存储格式。

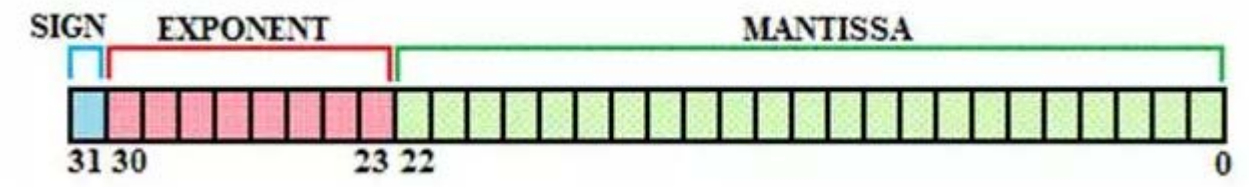

图 3.4 单精度浮点数

图 3.5 是半精度浮点数(float16)存储格式。

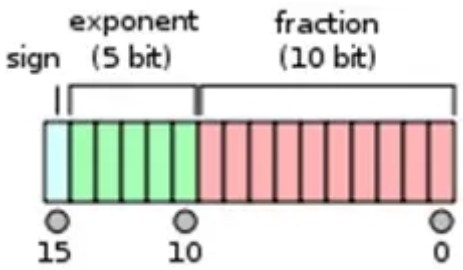

图 3.5 半精度浮点数

浮点数存储格式分成 3 部分，符号位，指数和尾数。不同精度是指数位和尾数位的长度不一样。

3.2 设备内存管理

TF 设备内存管理模块利用 BFC 算法（best-fit with coalescing）实现。BFC 算法是 Doung Lea’s malloc(dlmalloc)的一个非常简单的版本。它具有内存分配、释放、碎片管理等基本功能[11]。

BFC 将内存分成一系列内存块，每个内存块由一个 chunk 数据结构管理。从 chunk 结构中可以获取到内存块的使用状态、大小、数据的基址、前驱和后继 chunk 等信息。整个内存可以通过一个 chunk 的双链表结构来表示。

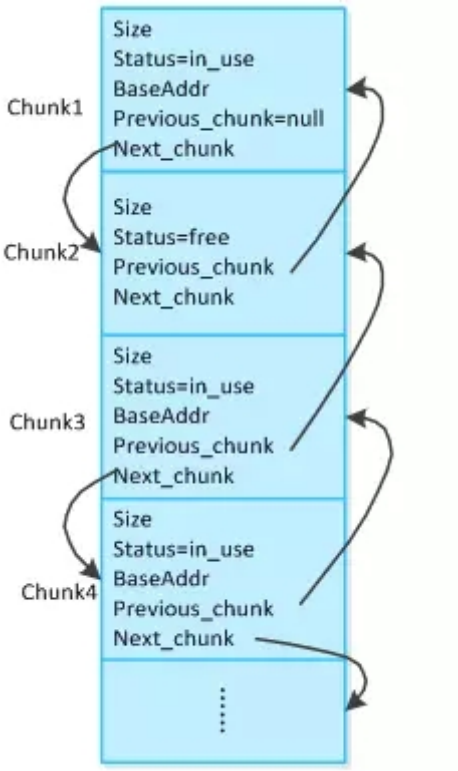

图 3.6 内存分块结构图

用户申请一个内存块（malloc）。根据建立的 chunk 双链表找到一个合适的内存块（后面会说明什么是合适的内存块），如果该内存块的大小是用户申请大小的两倍以上，那么将该内存块切分成两块，这就是 split 操作。返回其中一块给用户，并将该内存块标识为占用。Spilt 操作会新增一个 chunk，所以需要修改 chunk 双链表以维持前驱和后继关系。

用户释放一个内存块（free）。先将该块标记为空闲。然后根据 chunk 数据结构中的信息找到其前驱和后继内存块。如果前驱和后继块中有空闲的块，那么将刚释放的块和空闲的块合并成一个更大的 chunk（这就是 merge 操作，合并当前块和其前后的空闲块）。再修改双链表结构以维持前驱后继关系。这就做到了内存碎片的回收。

BFC 的核心思想是：将内存分块管理，按块进行空间分配和释放；通过 split 操作将大内存块分解成小内存块；通过 merge 操作合并小的内存块，做到内存碎片回收。

但是还留下许多疑问。比如说申请内存空间时，什么样的块算合适的内存块？如何快速管理这种块？

BFC 算法采取的是被动分块的策略。最开始整个内存是一个 chunk，随着用户申请空间的次数增加，最开始的大 chunk 会被不断的 split 开来，从而产生越来越多的小 chunk。当 chunk 数量很大时，为了寻找一个合适的内存块而遍历双链表无疑是一笔巨大的开销。为了实现对空闲块的高效管理，BFC 算法设计了 bin 这个抽象数据结构。

Bin 数据结构中，每个 bin 都有一个 size 属性，一个 bin 是一个拥有 chunk size >= bin size 的空闲 chunk 的集合。集合中的 chunk 按照 chunk size 的升序组织成单链表。BFC 算法维护了一个 bin 的集合：bins。它由多个 bin 以及从属于每个 bin 的 chunks 组成。内存中所有的空闲 chunk 都由 bins 管理。

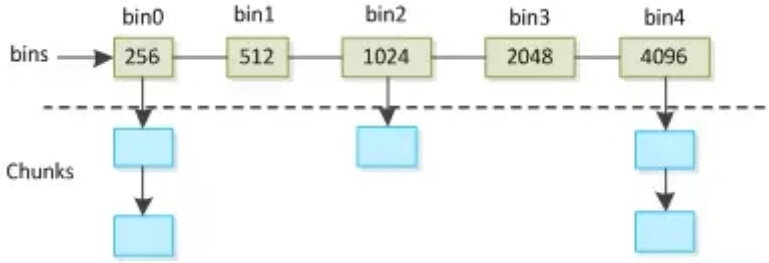

图 3.7 bins 集合的结构图

图 3.7 中每一列表示一个 bin，列首方格中的数字表示 bin 的 size。bin size 的大小都是 256 的 2^n 的倍。每个 bin 下面挂载了一系列的空闲 chunk，每个 chunk 的 chunk size 都大于等于所属的 bin 的 bin size，按照 chunk size 的升序挂载成单链表。BFC 算法针对 bins 这个集合设计了三个操作：search、insert、delete。

Search 操作：给定一个 chunk size，从 bins 中找到大于等于该 chunk size 的最小的那个空闲 chunk。Search 操作具体流程如下。如果 bin 以数组的形式组织，那么可以从 index = chunk size /256 >>2 的那个 bin 开始查找。较好的情况是开始查找的那个 bin 的 chunk 链表非空，那么直接返回链表头即可。这种情况时间复杂度是常数级的。最坏的情况是遍历 bins 数组中所有的 bin。对于一般大小的内存来说，bins 数组元素非常少，比如 4G 空间只需要 23 个 bin 就足够了（256 * 2 ^ 23 > 4G），因此也很快能返回结果。总体来说 search 操作是非常高效的。对于固定大小内存来说，查找时间是常数量级的。

Insert 操作：将一个空闲的 chunk 插入到一个 bin 所挂载的 chunk 链表中，同时需要维持 chunk 链表的升序关系。具体流程是直接将 chunk 插入到 index = chunk size /256 >>2 的

那个 bin 中即可。

Delete 操作：将一个空闲的 chunk 从 bins 中移除。

TF 中内存分配算法实现文件 core/common_runtime/bfc_allocator.cc，GPU 内存分配算法实现文件 core/common_runtime/gpu/gpu_bfc_allocator.cc。

3.3 TF 开发工具介绍

TF 系统开发使用了 bazel 工具实现工程代码自动化管理，使用了 protobuf 实现了跨设备数据传输，使用了 swig 库实现 python 接口封装。本章将从这三方面介绍 TF 开发工具的使用。

3.3.1 Swig 封装

Tensorflow 核心框架使用 C++编写，API 接口文件定义在 tensorflow/core/public 目录下，主要文件是 tensor_c_api.h 文件，C++语言直接调用这些头文件即可。

Python 通过 Swig 工具封装 TF 库包间接调用，接口定义文件 tensorflow/python/tensorflow.i。其中 swig 全称为 Simplified Wrapper and Interface Generator，是封装 C/C++并与其它各种高级编程语言进行嵌入联接的开发工具，对 swig 感兴趣的请参考相关文档。

在 tensorflow.i 文件中包含了若干个.i 文件，每个文件是对应模块的封装，其中 tf_session.i 文件中包含了 tensor_c_api.h，实现 client 向 session 发送请求创建和运行 graph 的功能。

3.3.2 Bazel 编译和调试

Bazel 是 Google 开源的自动化构建工具，类似于 Make 和 CMake 工具。Bazel 的目标是构建“快速并可靠的代码”，并且能“随着公司的成长持续调整其软件开发实践”。

TF 中几乎所有代码编译生成都是依赖 Bazel 完成的，了解 Bazel 有助于进一步学习 TF 代码，尤其是编译测试用例进行 gdb 调试。

Bazel 假定每个目录为[package]单元，目录里面包含了源文件和一个描述文件 BUILD，描述文件中指定了如何将源文件转换成构建的输出。

以图 3.8 为例，左子图为工程中不同模块间的依赖关系，右子图是对应模块依赖关系的 BUILD 描述文件。

图 3.8 中 name 属性来命名规则，srcs 属性为模块相关源文件列表，deps 属性来描述规则之间的依赖关系。”//search: google_search_page”中”search”是包名，”google_search_page”为规则名，其中冒号用来分隔包名和规则名；如果某条规则所依赖的规则在其他目录下，就用"//"开头，如果在同一目录下，可以忽略包名而用冒号开头。

图 3.8 中 cc_binary 表示编译目标是生成可执行文件，cc_library 表示编译目标是生成库文件。如果要生成 google_search_page 规则可运行

```
$ bazel build –c opt //search:google_search_page
```

如果要生成可调试的二进制文件，可运行

```
$ bazel build –c dbg //search:google_search_page
```

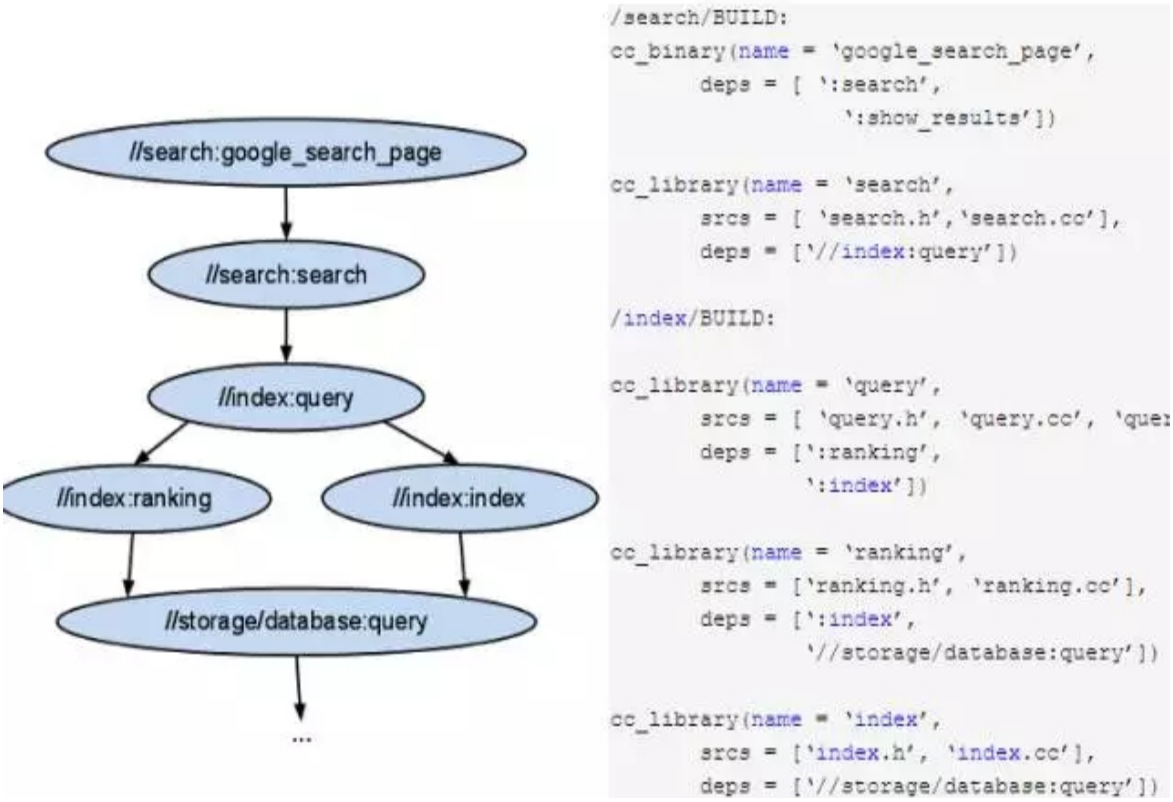

图 2.8 Bazel BUILD 文件示例

TF 中首次运行 bazel 时会自动下载很多依赖包，如果有的包下载失败，打开 tensorflow/workspace.bzl 查看是哪个包下载失败，更改对应依赖包的 new_http_archive 中的 url 地址，也可以把 new_http_archive 设置为本地目录 new_local_repository。

TF 中测试用例跟相应代码文件放在一起，如 MatMul 操作的 core/kernels/matmul_op.cc 文件对应的测试用例文件为 core/kernels/matmul_op_test.cc 文件。运行这个测试用例需要查找这个测试用例对应的 BUILD 文件和对应的命令规则，如 matmul_op_test.cc 文件对应的 BUILD 文件为 core/kernels/BUILD 文件，如下

```
tf_cuda_cc_test(
    name = "matmul_op_test",
    size = "small",
    deps = [
        ":matmul_op",
        ":ops_testutil",
        ":ops_util",
        "//tensorflow/core:core_cpu",
        "//tensorflow/core:framework",
        "//tensorflow/core:lib",
        "//tensorflow/core:protos_all_cc",
        "//tensorflow/core:test",
        "//tensorflow/core:test_main",
        "//tensorflow/core:testlib",
    ],
)
```

其中 tf_cuda_cc_test 函数是 TF 中自定义的编译函数，函数定义在/tensorflow/tensorflow.bzl 文件中，它会把 matmul_op_test.cc 放进编译文件中。要生成 matmul_op_test 可执行文件可运行如下脚本：

```
$ bazel build -c dbg //tensorflow/core/kernels:matmul_op_test
$ gdb bazel-bin/tensorflow/core/kernels/matmul_op_test
```

3.3.3 Protobuf 序列化

Protocol Buffers 是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。它很适合做数据存储或 RPC 数据交换格式。可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。

Protobuf 对象描述文件为.proto 类型，编译后生成.pb.h 和.pb.cc 文件。

Protobuf 主要包含读写两个函数：Writer（序列化）函数 SerializeToOstream()和 Reader（反序列化）函数 ParseFromIstream()。

Tensorflow 在 core/probobuf 目录中定义了若干与分布式环境相关的.proto 文件，同时在

core/framework 目录下定义了与基本数据类型和结构的.proto 文件，在 core/util 目录中也定义部分.proto 文件，感觉太随意了。

在分布式环境中，不仅需要传输数据序列化，还需要数据传输协议。Protobuf 在序列化处理后，由 gRPC 完成数据传输。gRPC 数据传输架构图见图 3.9。

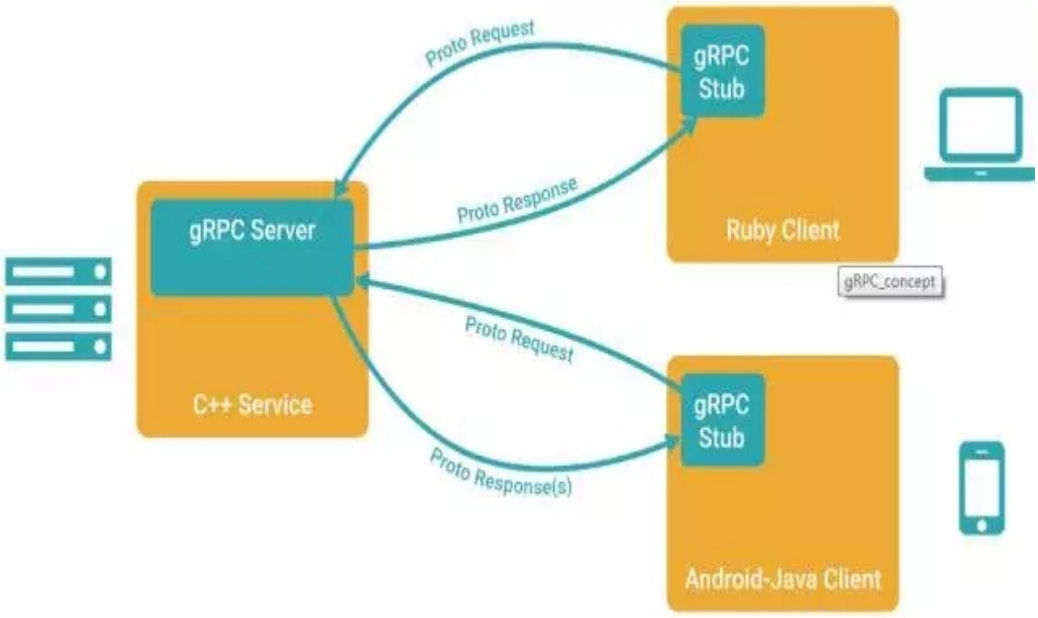

图 3.9 gRPC 数据传输架构

gRPC 服务包含客户端和服务端。gRPC 客户端调用 stub 对象将请求用 protobuf 方式序列化成字节流，用于线上传输，到 server 端后调用真正的实现对象处理。gRPC 的服务端通过 observer 观察处理返回和关闭通道。
TF 使用 gRPC 完成不同设备间的数据传输，比如超参数、梯度值、graph 结构。

4. TF-Kernels 模块

TF 中包含大量 Op 算子，这些算子组成 Graph 的节点集合。这些算子对 Tensor 实现相应的运算操作。图 4.1 列出了 TF 中的 Op 算子的分类和举例。

Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural-net building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

图 4.1 TensorFlow 核心库中的部分运算

4.1 OpKernels 简介

OpKernel 类（core/framework/op_kernel.h）是所有 Op 类的基类。继承 OpKernel 还可以自定义新的 Op 类。用的较多的 Op 如（MatMul, Conv2D, SoftMax, AvgPooling, Argmax 等）。

所有 Op 包含注册（Register Op）和实现（正向计算、梯度定义）两部分。

所有 Op 类的实现需要 overide 抽象基函数 void Compute(OpKernelContext* context)，实现自身 Op 功能。用户可以根据需要自定义新的 Op 操作，参考[12]。

TF 中所有 Op 操作的属性定义和描述都在 ops/ops.pbtxt。如下 Add 操作，定义了输入参数 x、y，输出参数 z。

```
op {
  name: "Add"
  input_arg {
    name: "x"      type_attr: "T"
  }
  input_arg {
    name: "y"      type_attr: "T"
  }
  output_arg {
    name: "z"      type_attr: "T"
  }
  attr {
    name: "T"      type: "type"
    allowed_values {
      list {
        type: DT_FLOAT
        ...
      }
    }
  }
  summary: "Returns x + y element-wise."
  description: "*NOTE*: Add supports broadcasting. AddN does not."
}
```

4.2 UnaryOp & BinaryOp

UnaryOp 和 BinaryOp 定义了简单的一元操作和二元操作，类定义在/core/kernels/cwise_ops.h 文件，类实现在/core/kernels/cwise_op_*.cc 类型的文件中，如 cwise_op_sin.cc 文件。

一元操作全称为 Coefficient-wise unary operations，一元运算有 abs， sqrt， exp， sin，cos， conj（共轭）等。如 abs 的基本定义：

```
template<typename T>
struct abs : base<T, Eigen::internal::scalar_abs_op<T>, typename Eigen::internal::scalar_abs_
op<T>::result_type> {};
```

二元操作全称为 Coefficient-wise binary operations，二元运算有 add，sub， div， mul，mod 等。如 sum 的基本定义：

```
template<typename T>
struct add : base<T, Eigen::internal::scalar_sum_op<T> > {  static const bool use_bcast_optim
ization = true; };
```

4.3 MatMul

4.3.1 Python 相关部分

在 Python 脚本中定义 matmul 运算：

```
tf.matmul(a, b, transpose_a=False,...)
```

根据 Ops 名称 MatMul 从 Ops 库中找出对应 Ops 类型
创建 ops 节点

```
graph.create_op(op_type_name, inputs, output_types,..)
[ops/op_def_library.py]
```

创建 ops 节点并指定相关属性和设备分配

```
node_def = _NodeDef(op_type, name, device=None, attrs=attrs)
ret = Operation(node_def, self, inputs=inputs,…)
[framework/ops.py]
```

4.3.2 C++相关部分

Python 脚本通过 swig 调用进入 C 接口 API 文件 core/client/tensor_c_api.cc，调用 TF_NewNode 函数生成节点，同时还需要指定输入变量，TF_AddInput 函数设置 first 输入变量，TF_AddInputList 函数设置 other 输入变量。这里 op_type 为 MatMul，first 输入变量为 a，other 输入变量为 b。

```
TF_NodeDescription* TF_NewNode(TF_Graph* graph, const char*
op_type,const char* node_name);
```
[public/tensor_c_api.h]

创建节点根据节点类型从注册的 Ops 工厂中生成，即 TF 通过工厂模式把一系列 Ops 注册到 Ops 工厂中。其中 MatMul 的注册函数为如下

```
REGISTER_OP("MatMul")
    .Input("a: T").Input("b: T").Output("product: T")
    .Attr("transpose_a: bool = false").Attr("transpose_b: bool = false")
    .Attr("T: {half, float, double, int32, complex64, complex128}")
    .SetShapeFn(shape_inference::MatMulShape).Doc(R"doc");
```
[ops/math_ops.cc]

4.3.3 MatMul 正向计算

MatMul 的实现部分在 core/kernels/matmul_op.cc 文件中，类 MatMulOp 继承于 OpKernel，成员函数 Compute 完成计算操作。

```
class MatMulOp : public OpKernel {…
        void Compute(OpKernelContext* ctx) override{}
…}
```

MatMul 的测试用例 core/kernels/matmul_op_test.cc 文件，要调试这个测试用例，可通过如下方式：

```
$ bazel build -c dbg //tensorflow/core/kernels:matmul_op_test
$ gdb bazel-bin/tensorflow/core/kernels/matmul_op_test
```

在 TF 中 MatMul 实现了 CPU 和 GPU 两个版本，其中 CPU 版本使用 Eigen 库，GPU 版本使用 cuBLAS 库。

CPU 版的 MatMul 使用 Eigen 库，调用方式如下：

```
functor::MatMulFunctor<CPUDevice,T>()(ctx->eigen_device<CPUDevice>(),out->
matrix<T>(), a.matrix<T>(),b.matrix<T>(), dim_pair);
```
[kernels/matmul_op.cc]

简而言之就是调用 eigen 的 constract 函数。

GPU 版的 MatMul 使用 cuBLAS 库，准确而言是基于 cuBLAS 的 stream_executor 库。Stream executor 是 google 开发的开源并行计算库，调用方式如下：

```
auto* stream = ctx->op_device_context()->stream();
auto a_ptr = AsDeviceMemory(a.template flat<T>().data());
stream->ThenBlasGemm(..., a_ptr, b_ptr, &c_ptr....)
```

[kernels/**matmul_op.cc**]

其中 stream 类似于设备句柄，可以调用 stream executor 中的 cuda 模块完成运算。

4.3.4 MatMul 梯度计算

MatMul 的梯度计算本质上也是一种 kernel ops，描述为 MatMulGrad。MatMulgrad 操作是定义在 grad_ops 工厂中，类似于 ops 工厂。定义方式如下：

```
REGISTER_OP_GRADIENT("MatMul", MatMulGrad);
```

[ops/**math_grad.cc**]

MatmulGrad 由 FDH（Function Define Helper）完成定义，

```
*g = FDH::Define(
    // Arg defs
    {"x: T", "y: T", "dz: T"},
    // Ret val defs
    {"dx: T", "dy: T"},
    // Attr defs
    {{"T: {half, float, double}"}},
    // Nodes
    {
        {{"dx"}, "MatMul", {"dz", "y"},
         {{"T", "$T"}, {attr_adj_x, ax0}, {attr_adj_y, ax1}}},
        {{"dy"}, "MatMul",{"x", "dz"},
         {{"T", "$T"}, {attr_adj_x, ay0}, {attr_adj_y, ay1}}},
    });
```

[ops/math_grad.cc]

其中 attr_adj_x="transpose_a" ax0=false, ax1=true, attr_adj_y= "transpose_b", ay0=true, ay1=false, *g 属于 FunctionDef 类，包含 MatMul 的梯度定义。

从 FDH 定义中可以看出 MatMulGrad 本质上还是 MatMul 操作。在矩阵求导运算中：

```
假设 z=MatMul(x, y)，则：
dx = MatMul(dz, y)，  dy = MatMul(x, dz)
```

MatMulGrad 的测试用例 core/ops/math_grad_test.cc 文件，要调试这个测试用例，可通过如下方式：

```
$ bazel build -c dbg //tensorflow/core:ops_math_grad_test
$ gdb bazel-bin/tensorflow/core/ops_math_grad_test
```

4.4 Conv2d

关于 conv2d 的 python 调用部分和 C++创建部分可参考 MatMul 中的描述。

4.4.1 Conv2d 正向计算部分

TF 中 conv2d 接口如下所示，简单易用：

实现部分在 core/kernels/conv_ops.cc 文件中，类 Conv2DOp 继承于抽象基类 OpKernel。

Conv2DOp 的测试用例 core/kernels/eigen_spatial_convolutions_test.cc 文件，要调试这个测试用例，可通过如下方式：

```
$ bazel build -c dbg //tensorflow/core/kernels:eigen_spatial_convolutions_test
$ gdb bazel-bin/tensorflow/core/kernels/eigen_spatial_convolutions_test
```

Conv2DOp 的成员函数 Compute 完成计算操作。

为方便描述，假设 tf.nn.conv2d 中 input 参数的 shape 为[batch, in_rows, in_cols, in_depth]，filter 参数的 shape 为[filter_rows, filter_cols, in_depth, out_depth]。

首先，计算卷积运算后输出 tensor 的 shape。

Ø 若 padding=VALID，output_size = (input_size - filter_size + stride) / stride;

Ø 若 padding=SAME，output_size = (input_size + stride - 1) / stride;

其次，根据计算结果给输出 tensor 分配内存。

然后，开始卷积计算。Conv2DOp 实现了 CPU 和 GPU 两种模式下的卷积运算。同时，还需要注意 input tensor 的输入格式，通常有 NHWC 和 NCHW 两种格式。在 TF 中，Conv2d-CPU 模式下目前仅支持 NHWC 格式，即[Number, Height, Weight, Channel]格式。Conv2d-GPU 模式下以 NCHW 为主，但支持将 NHWC 转换为 NCHW 求解。C++中多维数组是 row-major 顺序存储的，而 Eigen 默认是 col-major 顺序的，则 C++中[N, H, W, C]相当于 Eigen 中的[C, W, H, N]，即 dimention order 是相反的，需要特别注意。

Conv2d-CPU 模式下调用 Eigen 库函数。

```
Eigen::SpatialConvolution(input,filter,col_stride,
row_stride,padding)
[kernels/eigen_spatial_convolutions.h]
```

Eigen 库中卷积函数的详细代码参见图 4 2。

```
choose(
     Cond<internal::traits<Input>::Layout == ColMajor>(),
     kernel
         .reshape(kernel_dims)
         .contract(input
                     .extract_image_patches(
                        kernelRows, kernelCols, row_stride, col_stride,
                        row_in_stride, col_in_stride, padding_type)
                     .reshape(pre_contract_dims),
                 contract_dims)
         .reshape(post_contract_dims),
     input
         .extract_image_patches(kernelRows, kernelCols, row_stride,
col_stride,
                                row_in_stride, col_in_stride, padding_type)
         .reshape(pre_contract_dims)
         .contract(kernel.reshape(kernel_dims), contract_dims)
         .reshape(post_contract_dims));
    [kernels/eigen_spatial_convolutions.h]
```

图 4.2 Eigen 卷积运算的定义

- Ø Tensor:: extract_image_patches () 为卷积或池化操作抽取与 kernel size 一致的 image patches。该函数的定义在 eigen3/unsupported/Eigen/CXX11/src/Tensor/TensorBase.h 中，参考该目录下 ReadME.md。

- Ø Tensor:: extract_image_patches () 的输出与 input tensor 的 data layout 有关。设 input tensor 为 ColMajor 格式[NHWC]，则 image patches 输出为[batch, filter_index, filter_rows, filter_cols, in_depth]，并 reshape 为[batch * filter_index, filter_rows * filter_cols * in_depth]，而 kernels 维度为[filter_rows * filter_cols * in_depth, out_depth]，然后 kernels 矩阵乘 image patches 得到输出矩阵[batch * filter_index, out_depth]，并 reshape 为[batch, out_rows, out_cols, out_depth]。

Conv2d-GPU 模式下调用基于 cuDNN 的 stream_executor 库。若 input tensor 为 NHWC 格式的，则先转换为 NCHW 格式

```
functor::NHWCToNCHW(ctx, input,&transformed_input);
```
[kernels/conv_ops_gpu_3.cu.cc]

调用 cudnn 库实现卷积运算：

```
dynload::cudnnGetConvolutionForwardAlgorithm(...)
dynload::cudnnGetConvolutionForwardWorkspaceSize(...)
dynload::cudnnConvolutionForward(...)
```
[stream_executor/cuda/cuda_dnn.cc]

计算完成后再转换成 HHWC 格式的

```
functor::NCHWToNHWC(ctx, transformed_output,&output);
```
[kernels/conv_ops_gpu_3.cu.cc]

4.4.2 Conv2d 梯度计算部分

Conv2D 梯度计算公式，假设 output=Conv2d(input, filter)，则

```
grad(input) = grad(output)*[sigmoid(Conv2d(input, filter)) /sigmoid(input)]
grad(filter) = grad(output)*[sigmoid(Conv2d(input, filter)) /sigmoid(filter)]
```

Conv2D 梯度计算的测试用例 core/kernels/eigen_backward_spatial_convolutions_test.cc 文件，要调试这个测试用例，可通过如下方式：

Conv2d 的梯度计算函数描述为 Conv2DGrad。Conv2DGrad 操作定义在 grad_ops 工厂中。注册方式如下：

```
REGISTER_OP_GRADIENT("Conv2D", Conv2DGrad);
```
[ops/nn_grad.cc]

Conv2DGrad 由 FDH（Function Define Helper）完成定义，参见图 4.3。

```
*g = FDH::Define(
  // Arg defs
  {"input: T", "filter: T", "grad: T"},
  // Ret val defs
  {"input_grad: T", "filter_grad: T"},
  // Attr defs
  {"T: {float, double}",
   "strides: list(int)",
   "use_cudnn_on_gpu: bool = true",
   GetPaddingAttrString(),
   GetConvnetDataFormatAttrString()},
  // Nodes
  {
    {{"i_shape"}, "Shape", {"input"}, {{"T", "$T"}}},
    {{"input_grad"}, "Conv2DBackpropInput", {"i_shape", "filter", "grad"},
     /*Attrs=*/{{"T", "$T"},
```

图 4.3 Conv2DGrad 的函数定义

Conv2DGrad 梯度函数定义中依赖 Conv2DBackpropInput 和 Conv2DBackpropFilter 两种 Ops，二者均定义在 kernels/conv_grad_ops.cc 文件中。

Conv2DBackpropInputOp 和 Conv2DBackpropFilterOp 的实现分为 GPU 和 CPU 版本。

Conv2D 运算的 GPU 版实现定义在类 Conv2DSlowBackpropInputOp 和类 Conv2DSlowBackpropFilterOp 中。

Conv2D 运算的 CPU 版有两种实现形式，分别为 custom 模式和 fast 模式。Custom 模式基于贾扬清在 caffe 中的思路实现，相关类是 Conv2DCustomBackpropInputOp 和 Conv2DCustomBackpropFilterOp。Fast 模式基于 Eigen 计算库，由于在 GPU 下会出现 nvcc 编译超时，目前仅适用于 CPU 环境，相关类是 Conv2DFastBackpropInputOp 和 Conv2DFastBackpropFilterOp。

根据 Conv2DGrad 的函数定义，从代码分析 Conv2D-GPU 版的实现代码，即分析 Conv2DBackpropInput 和 Conv2DBackpropFilter 的实现方式。

```
REGISTER_KERNEL_BUILDER(Name("Conv2DBackpropInput")
        .Device(DEVICE_GPU).TypeConstraint<Eigen::half>("T")
        .HostMemory("input_sizes"),
        Conv2DSlowBackpropInputOp<GPUDevice, Eigen::half>);
REGISTER_KERNEL_BUILDER(Name("Conv2DBackpropFilter")
        .Device(DEVICE_GPU).TypeConstraint<Eigen::half>("T")
        .HostMemory("filter_sizes"),
        Conv2DSlowBackpropFilterOp<GPUDevice, Eigen::half>);
```

[kernels/conv_grad_ops.cc]

Conv2DSlowBackpropInputOp 的成员函数 Compute 完成计算操作。

Compute 实现部分调用 stream executor 的相关函数，需要先获取库的 stream 句柄，再调用卷积梯度函数。

```
Stream::ThenConvolveBackwardDataWithAlgorithm(filter_desc, filter_ptr,
  output_desc, out_backprop_ptr, conv_desc, input_desc, &in_backprop_ptr,
  &scratch_allocator, algorithm_config, nullptr))
```

[steam_executor/stream.cc]

stream executor 在卷积梯度运算部分仍然是借助 cudnn 库实现的。

```
CudnnSupport::DoConvolveBackwardDataImpl(
    stream, CUDNN_DATA_FLOAT, filter_descriptor, filter_data,
    output_descriptor_in, backward_output_data, convolution_descriptor,
    input_descriptor, backward_input_data, scratch_allocator,
    algorithm_config, output_profile_result);
```

[steam_executor/cuda/cuda_dnn.cc]

5. TF-Graph 模块

TF 把神经网络模型表达成一张拓扑结构的 Graph，Graph 中的一个节点表示一种计算算子。Graph 从输入到输出的 Tensor 数据流动完成了一个运算过程，这是对类似概率图、神经网络等连接式算法很好的表达，同时也是对 Tensor + Flow 的直观解释。

5.1 Graph 视图

Tensorflow 采用符号化编程，形式化为 Graph 计算图。Graph 包含节点（Node）、边（Edge）、NameScope、子图（SubGraph），图 5.1 是 Graph 的拓扑描述。

- 节点分为计算节点(Compute Node)、起始点（Source Node）、终止点（Sink Node）。起始点入度为 0，终止点出度为 0。
- NameScope 为节点创建层次化的名称，图 3 4 中的 NameSpace 类型节点就是其中一种体现。
- 边分为普通边和依赖边（Dependecy Edge）。依赖边表示对指定的计算节点有依赖性，必须等待指定的节点计算完成才能开始依赖边的计算。

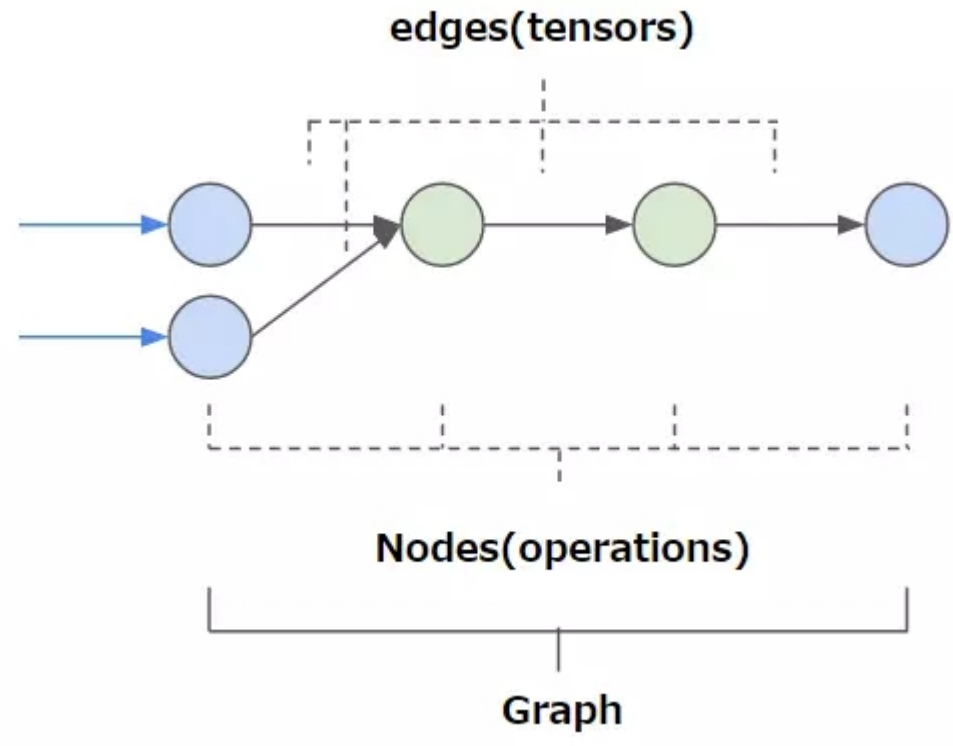

图 5.1 Graph 的拓扑描述

图 5.2 是 Graph 的 UML 视图模型，左侧 GraphDef 类为 protobuf 中定义的 graph 结构，可将 graph 结构序列化和反序列化处理，用于模型保存、模型加载、分布式数据传输。右侧 Graph 类为/core/graph 模块中定义的 graph 结构，完成 graph 相关操作，如构建(construct)，剪枝(pruning)、划分(partitioning)、优化(optimize)、运行(execute)等。GraphDef 类和 Graph 类可以相关转换，如图中中间部分描述，函数 Graph::ToGraphDef()将 Graph 转换为 GraphDef，函数 ConvertGraphDefToGraph 将 GraphDef 转换为 Graph，借助这种转换就能实现 Graph 结构的网络传输。

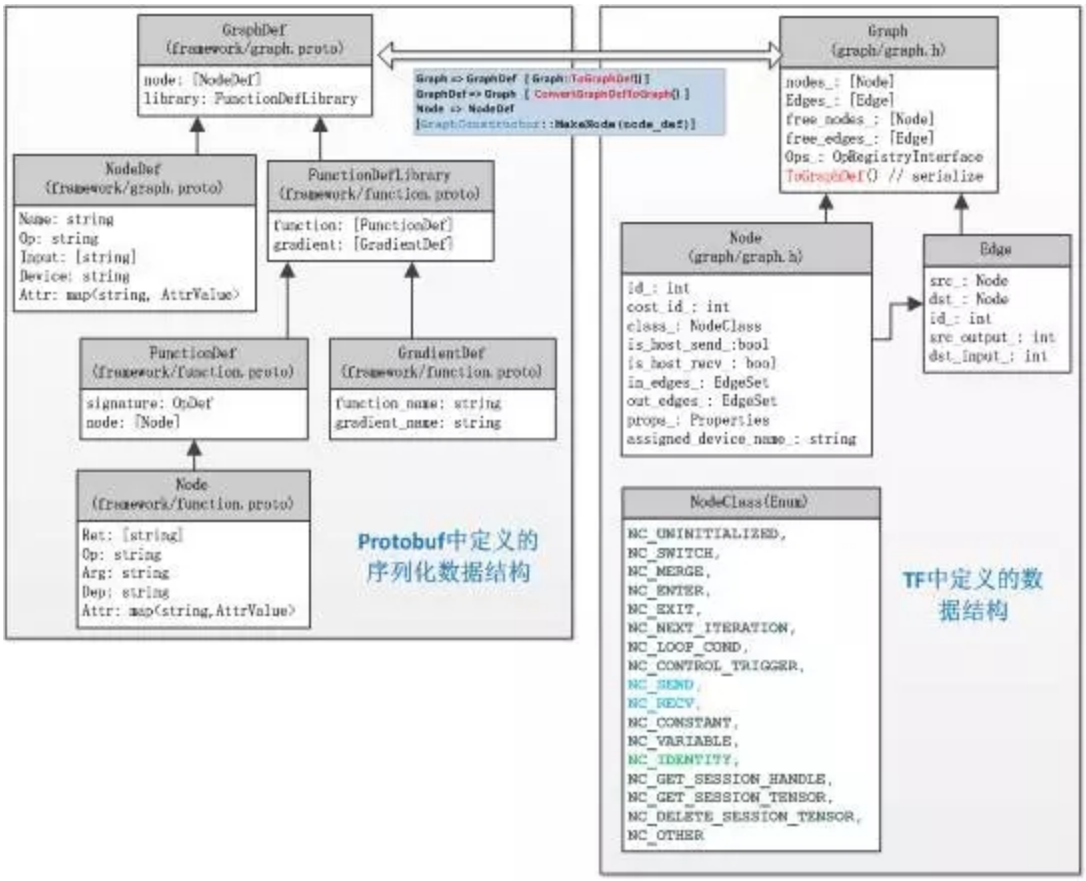

图 5.2 Graph 的 UML 视图

Graph-UML 图中还定义了 Node 和 Edge。Node 定义函数操作和属性信息，Edge 连接源节点和目标节点。类 NodeDef 中定义了 Op、Input、Device、Attr 信息，其中 Device 可能是 CPU、GPU 设备，甚至是 ARM 架构的设备，说明 Node 是与设备绑定的。类 FunctionDefLibrary 主要是为了描述各种 Op 的运算，包括 Op 的正向计算和梯度计算。FunctionDef 的定义描述见图 5.3。

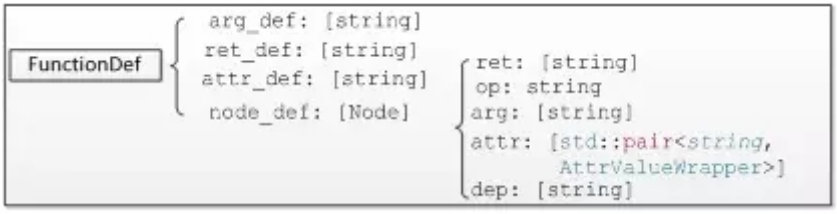

图 5.3 FunctionDef 的定义

图 5.4 是 FunctionDef 举例，对 MatMulGrad 的梯度描述，其中包含函数参数定义、函数返回值定义、模板数据类型定义、节点计算逻辑。

```
*g = FDH::Define(
      // Arg defs
      {"x: T", "y: T", "dz: T"},
      // Ret val defs
      {"dx: T", "dy: T"},
      // Attr defs
      {{"T: {half, float, double}"}},
      // Nodes
      {
          {{"dx"}, "MatMul", {"dz", "y"},
           {{"T", "$T"}, {attr_adj_x, ax0}, {attr_adj_y, ax1}}},
          {{"dy"}, "MatMul",{"x", "dz"},
           {{"T", "$T"}, {attr_adj_x, ay0}, {attr_adj_y, ay1}}},
      });
其中: attr_adj_x="transpose_a" ax0=false, ax1=true,
    attr_adj_y="transpose_b", ay0=true, ay1=false
[ops/math_grad.cc]
```

图 5.4 FunctionDef 举例：MatMulGrad

5.2 Graph 构建

有向图（DAG）由节点和有向边组成。本章节主要讲述 TF 如何利用<Nodes, Edges>组合成完整的 graph 的。假设有如下计算表达式：t1=MatMul(input, W1)。

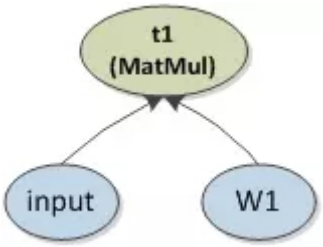

图 5.5 Graph 简单示例

图 5.5 中图计算表达式包含 3 个节点，2 条边，描述为字符串形式如下。

```
node { name: 'W1'    op: 'TestParams' },
node { name: 'input'    op: 'TestInput'    input: [ '^W1' ] },
node { name: 't1'    op: 'MatMul'    input: [ 'W1', 'input:1' ] }
```

TF先调用protobuf的解析方法将graph的字符串描述解析并生成GraphDef实例。

```
protobuf::TextFormat::ParseFromString(gdef_str, &gdef_)
```

然后将GraphDef实例转化为tensorflow::Graph实例，这个过程由tensorflow::GraphConstructor类完成。GraphConstructor先判别node的字符串格式是否正确，然后执行convert函数。

```
GraphConstructor::Convert()
```

[graph/graph_construtor.cc]

首先，按拓扑图的顺序逐步添加node和edge到graph中。

```
Graph::AddNode(const NodeDef& node_def, Status* status)
Graph::AddEdge(Node* source, int x, Node* dest, int y)
```

[graph/graph.cc]

然后，找出所有起始点（source node）和终止点（sink node）。

```
FixupSourceAndSinkEdges(Graph* g)
```

[graph/algorithms.cc]

接着，对graph进行优化。图优化部分请参考章节6.5。

```
OptimizeCSE(Graph* g, std::function<bool(const Node*)> consider_fn);
```

[graph/optimizer_cse.cc]

TF的graph构建模块测试用例在core/graph/graph_constructor_test.cc文件中。

```
$ bazel build -c dbg //tensorflow/core/graph_graph_constructor_test
```

5.3 Graph 局部执行

Graph 的局部执行特性允许使用者从任意一个节点输入（feed），并指定目标输出节点（fetch）。图 5.6 是 TF 白皮书中描述 Graph 局部执行的图。

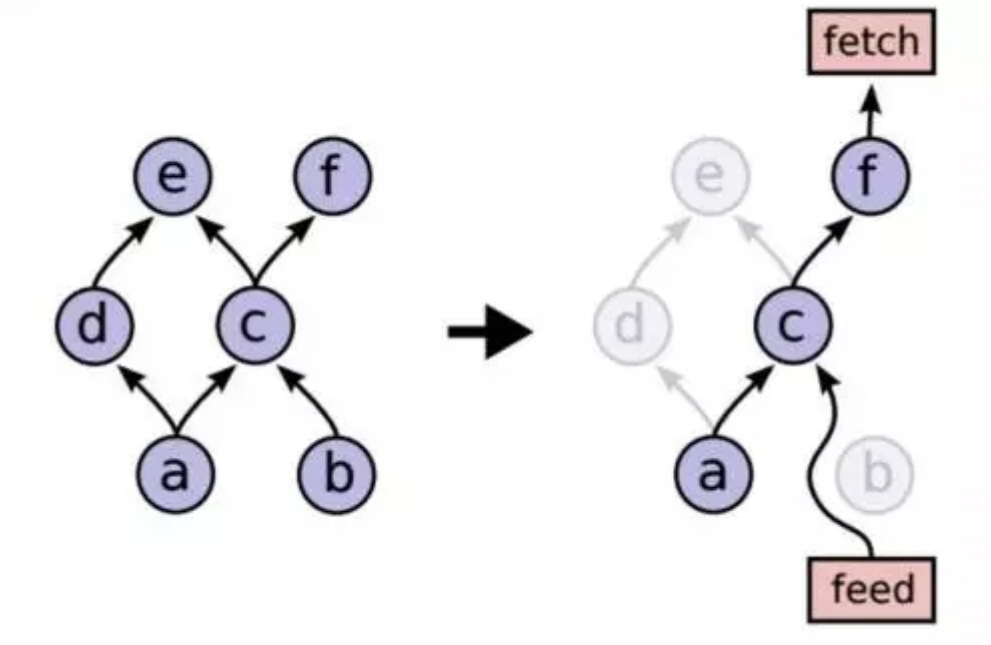

图 5.6 Graph 局部执行

图 5 6中左侧为计算图，如果要实现f=F(c)运算，代码如下：

```
result=sess.run(f, feed_dict={c: input})
```

TF是如何知道两个点之间的计算路径呢？这里涉及传递闭包的概念。传递闭包就是根据graph中节点集合和有向边的集合，找出从节点A到节点B的最小传递关系。如上图中，点a到点f的传递闭包是a -> c -> f。

Graph局部执行过程就是找到feed和fetch的最小传递闭包，这个传递闭包相当于原graph的subgraph。代码文件在graph/subgraph.cc中，函数RewriteGraphForExecution()在确定feed节点和fetch节点后，通过剪枝得到最小传递子图。

```
PruneForTargets(g, name_index, fetch_nodes, target_node_names));
[graph/subgraph.cc]
```

剪枝操作的实现函数如下，Graph通过模拟计算流标记出节点是否被访问，剔除未被访问的节点。

```
PruneForReverseReachability(Graph* g, std::unordered_set<const Node*> visited)
[graph/algorithm.cc]
```

5.4 Graph 设备分配

TF 具有高度设备兼容性，支持 X86 和 Arm 架构，支持 CPU、GPU 运算，可运行于 Linux、MacOS、Android 和 IOS 系统。而且，TF 的设备无关性特征在多设备分布式运行上也非常有用。

Graph 中每个节点都分配有设备编号，表示该节点在相应设备上完成计算操作。用户既可以手动指定节点设备，也可以利用 TF 自动分配算法完成节点设备分配。设备自动算法需要权衡数据传输代价和计算设备的平衡，尽可能充分利用计算设备，减少数据传输代价，从而提高计算性能。

Graph 设备分配用于管理多设备分布式运行时，哪些节点运行在哪个设备上。TF 设备分配算法有两种实现算法，第一种是简单布放算法（Simple Placer），第二种基于代价模型（Cost Model）评估。简单布放算法按照指定规则布放，比较简单粗放，是早期版本的 TF 使用的模型，并逐步被代价模型方法代替。

5.4.1 Simple Placer 算法

TF 实现的 Simple Placer 设备分配算法使用 union-find 方法和启发式方法将部分不相交且待分配设备的 Op 节点集合合并，并分配到合适的设备上。

Union-find（联合-查找）算法是并查集数据结构一种应用。并查集是一种树型的数据结构，其保持着用于处理一些不相交集合（Disjoint Sets）的合并及查询问题。Union-find 定义了两种基本操作：Union 和 Find。

- Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
- Union：将两个子集合并成同一个集合。即将一个集合的根节点的父指针指向另一个集合的根节点。

启发式算法（Heuristic Algorithm）定义了节点分配的基本规则。Simple Placer 算法默认将起始点和终止点分配给 CPU，其他节点中 GPU 的分配优先级高于 CPU，且默认分配给 GPU:0。启发式规则适用于以下两种场景：

- 对于符合 GeneratorNode 条件（0-indegree, 1-outdegree, not ref-type）的节点，让 node 与 target_node 所在 device 一致，参见图 5.7。

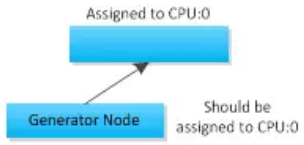

图 5 7 启发式规则A

Ø 对于符合MetaDataNode条件（即直接在原数据上的操作，如reshape）的节点，让node与source_node所在device一致，参见图 5 8。

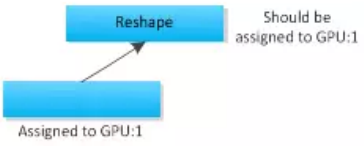

图 5 8 启发式规则B

TF 中 Simple Placer 的实现定义在文件 core/common_runtime/simple_placer.cc。文件中主要定义了两个类:ColocationGraph 和 SimplePlacer。ColocationGraph 类利用 Union-find 算法将节点子集合合并成一个节点集合，参考成员函数 ColocationGraph:: ColocateNodes 实现。SimplePlacer 类实现节点分配过程，下面将主要介绍 SimplePlacer:: Run()函数的实现过程。

```
SimplePlacer::Run()
```

首先，将graph中的node加入到ColocationGraph实例中，不包含起始点和终止点。

```
ColocationGraph colocation_graph(graph_, devices_, options_);
colocation_graph.AddNode(*node); [for node in _graph]
```

然后，找出graph中受constraint的edge(即src_node被指定了device的edge)，强制将dst_node指定到src_node所在的device。

```
colocation_graph.ColocateNodes(*edge->src(), *node);
```

最后，根据graph中已有的constraint条件为每个no-constraint的node指定device。

```
if IsGeneratorNode(node)    AssignAndLog(assigned_device, node);
if IsMetadataNode(node)     AssignAndLog(assigned_device, node);
```

Simple Placer的测试用例core/common_runtime/simple_placer_test.cc文件，要调试这个测试用例，可通过如下方式：

```
$ bazel build -c dbg //tensorflow/core:common_runtime_simple_placer_test
$ gdb bazel-bin/tensorflow/core/common_runtime_simple_placer_test
```

5.4.2 代价模型

TF 使用代价模型（Cost Model）会在计算流图生成的时候模拟每个 device 上的负载，并利用启发式策略估计 device 上的完成时间，最终找出预估时间较低的 graph 设备分配方案。Cost model 预估时间的方法有两种：

- 使用启发式的算法，通过把输入和输出的类型以及 tensor 的大小输入进去，得到时间的预估
- 使用模拟的方法，对图的计算进行一个模拟，得到各个计算在其可用的设备上的时间。

启发式策略会根据如下数据调整 device 的分配：节点任务执行的总时间；单个节点任务执行的累计时间；单个节点输出数据的尺寸。

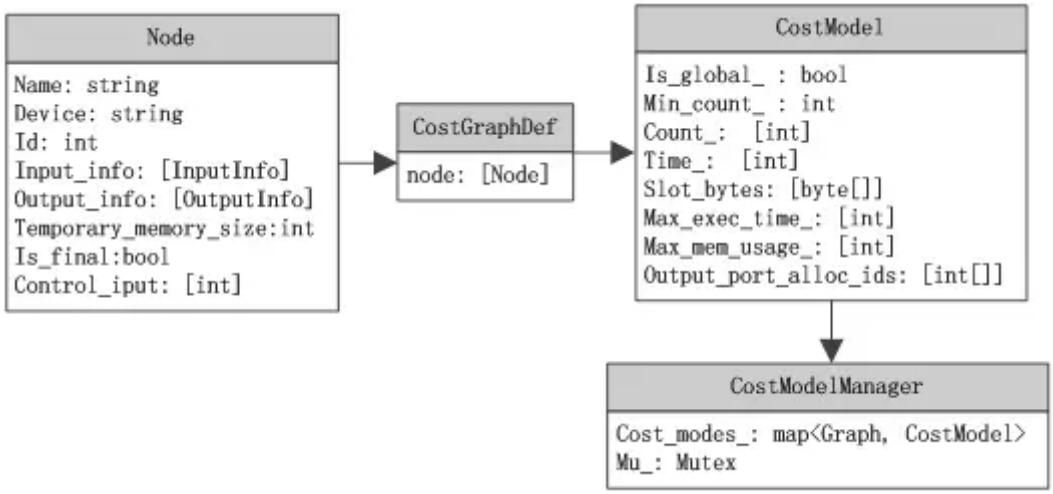

图 5.9 代价模型 UML 视图

TF 中代价模型的实现定义在文件 core/graph/costmodel.cc 和 core/common_runtime/costmodel_manager.cc，其 UML 视图参见图 5.9。

Cost model manager 从 graph 创建 cost model，再评估计算时间，如下。

```
void CostModel::InitFromGraph(const Graph& g) {
  AddNodesToCostModel(g, this);
  AssignSizes(g, this);
  EstimateComputationCosts(g, this);
  CheckInitialized(g);
}
```

其中评估时间的函数EstimateComputationCosts是对graph中每个node依次评估，节点计算时间评估函数如下。

```
TimeEstimateForNode(CostModel* cost_model, Node* n)
```

5.5 Graph优化

Graph优化算法利用一些优化策略，降低graph的计算复杂度和空间复杂度，提高graph运行速度。

Graph优化算法的实现在文件core/common_runtime/graph_optimizer.cc。

```
GraphOptimizer::Optimize(FunctionLibraryRuntime* runtime,
                         Device* device, Graph** graph)
```

Graph优化策略有三种：

Ø Common Subexpression Elimination (CSE, 公共子表达式消除)
如果一个表达式E已经计算过了，并且从先前的计算到现在的E中的变量都没有发生变化，那么E的此次出现就成为了公共子表达式。例如：x=(a+c)*12+(c+a)*2; 可优化为 x=E*14。

CSE实现函数如下，具体细节参考文献[16]。

```
OptimizeCSE(Graph* g, std::function<bool(const Node*)>
            consider_fn);
```

[graph/optimizer_cse.cc]

CSE测试用例在文件graph/optimizer_cse_test.cc中，调试方法：

```
$ bazel build -c dbg //tensorflow/core:graph_optimizer_cse_test
$ gdb bazel-bin/tensorflow/core/graph_optimizer_cse_test
```

Ø Constant Folding (常量合并)

在编译优化时，变量如果能够直接计算出结果，那么变量将有常量直接替换。例如：a=3+1-3*1; 可优化为a=1。

常量合并的实现函数如下。

```
DoConstantFolding(const ConstantFoldingOptions& opts,Device*
partition_device, Graph* g)
```

[common_runtime/constant_folding.cc]

常量合并的测试用例在common_runtime/constant_folding_test.cc中，调试方法：

```
$ bazel build -c dbg //tensorflow/core: common_runtime_constant_folding_test
$ gdb bazel-bin/tensorflow/core/common_runtime_constant_folding_test
```

Function Inlining (函数内联)

函数内联处理可减少方法调用的成本。在 TF 中包含以下几种方法：

- RemoveListArrayConverter(g)：” Rewrites _ListToArray and _ArrayToList to a set of Identity nodes”.
- RemoveDeadNodes(g)：删除 DeatNode。DeatNode 的特征是”not statefull, not _Arg, not reachable from _Retval”.
- RemoveIdentityNodes(g)：删除 Identity 节点。如 n2=Identity(n1) + Identity(n1); 优化后: n2=n1 + n1;
- FixupSourceAndSinkEdges(g)：固定 source 和 sink 的边
- ExpandInlineFunctions(runtime, g)：展开内联函数的嵌套调用

其中_ListToArray、_ArrayToList、_Arg、_Retval 均在 core/ops/function_ops.cc 中定义。

Graph 优化相关测试文件在 common_runtime/function_test.cc，调试方法：

```
$ bazel build -c dbg //tensorflow/core:common_runtime_function_test
$ gdb bazel-bin/tensorflow/core/common_runtime_function_test
```